CS1114 Lambda Abstractions Problem Set

April 10, 2025

Instructions:

- Every student, however, must submit his or her own submission for credit.
- Questions do not necessarily appear in order of difficulty
- Each solution should be thoroughly **designed**, following each step of the design recipe in detail.
- In this problem set, you will design functions in Racket using **lambda abstractions** (anonymous functions). Each problem requires you to use or reason about lambda expressions as part of your solution.
- You may use built-in higher-order functions like `map`, `filter`, etc., and you may define helper functions if needed.
- Make sure to use the exact function names, arities and argument order given so that the autograder can properly test your code.

1. **Function Composition (two functions).** Define a function `compose-2` that takes two arguments (two unary functions, f and g) and returns a new function. The returned function should take one argument x and produce $f(g(x))$. In other words, `compose-2` should **compose** f after g. Use a lambda to create the returned composed function.
   - *Example:* Let f be a function that adds 1 to its input (so `(f 5)` $= 6$), and let g be a function that doubles its input (so `(g 5)` $= 10$). Then `(compose-2 f g)` returns a function that, when applied to 5, yields `(f (g 5)) = (f 10) = 11`.
   - *Example (continued):* In code, if `(compose-2 f g)` produces a function `h`, then `(h 5)` should give the same result as `(f (g 5))`.
2. **Compose a List of Functions.** Define a function `compose-n` that takes one argument (a list of unary functions) and returns a new function that is the composition of all those functions. The returned function should take a single argument and apply each function from the list in sequence to produce the result. If the list is empty, the composed function should simply return its input (the identity function).

   Use `foldr` to build the composed function. The base case (`foldr`'s initial accumulator) should be the identity function (a function that returns its argument). At each step, use a lambda to compose the next function with the accumulated function. *Hint:* Your `foldr` combining function can have the form `(lambda (fun acc) (lambda (x) ...))`, where you fill in the body to call `fun` and `acc` appropriately.

Once `compose-n` returns the composed function, you can test it by applying that function to a value. For example:

- *Example:* Suppose we have three functions: `f1` (doubles a number), `f2` (negates a number), and `f3` (adds 10 to a number). If we create `H = (compose-n (list f1 f2 f3))`, then for any input `x`, `(H x)` produces the same result as `(f1 (f2 (f3 x)))`. For instance, if the input is 5, then `(f3 5)` = 15, `(f2 15)` = -15, and `(f1 -15)` = -30. Thus, `(H 5)` evaluates to -30.
- *Example:* If the list contains two functions [`sub1`, `add1`] (subtract 1 and add 1), the composed function should behave like the identity function. For instance, `((compose-n (list sub1 add1)) 10)` returns 10.

3. Design a function `repeat-func` that consumes two arguments: a one-argument function f and a nonnegative integer k. `repeat-func` should return a new one-argument **function** that, when applied to some value x, returns the result of applying f to x exactly k times in succession. The returned function must be constructed using a lambda.

   - If $k = 0$, the returned function should simply return x (i.e. act as the identity function on x).
   - If $k = 3$ and f is a function that adds 1 to its input, then `((repeat-func f 3) 5)` should yield `8` (since applying the $+1$ function three times to 5 gives 8).
   - If $k = 1$, `(repeat-func f 1)` should return a function equivalent to f itself (so `((repeat-func f 1) x)` produces the same result as `(f x)` for any x).

   Your design should handle any $k \geq 0$. Consider using recursion or iteration in your solution, and ensure that you use a lambda to create the returned function. (Hint: You may need to use a lambda for the base case and/or recursively build the composed function.)

4. Design a function `all-pred` that consumes a list of one-argument **predicate** functions (each predicate returns `#t` or `#f` when applied to a value). `all-pred` should return a new one-argument **predicate function** that takes an input x and returns true if and only if **every** predicate in the list returns true when given x. The returned combined predicate must be constructed as a lambda.

   - For example, suppose `pred-list` is `(list even? positive?)` (where `even?` and `positive?` are built-in predicates for numbers). Then `(define all-even-pos (all-pred pred-list))` produces a predicate function `all-even-pos`.
   - `(all-even-pos 4)` should produce `#t` (since 4 is both even and positive).
   - `(all-even-pos -2)` should produce `#f` (since -2 is even but not positive).
   - `(all-even-pos 3)` should produce `#f` (since 3 is positive but not even).
   - If the list of predicates is empty, the combined predicate should always return `#t` for any input (since there are no conditions to fail).

   Ensure that your `all-pred` function uses at least one lambda in constructing the result. You may find higher-order functions like `andmap` helpful, but you can also solve it with explicit recursion or `foldr`. In any case, make sure the final returned function is an anonymous lambda.

5. Design a function `make-censor` that helps create a censorship filter for strings. `make-censor` consumes a string of forbidden characters (each character in this string is one that should be censored out). It should return a **function** that consumes a text string and returns a new string in which every character that appears in the forbidden list is replaced with an underscore _, while all other characters remain the same. The returned string-processing function should be constructed using a lambda.

   - For example, `(define censor-vowels (make-censor "aeiou"))` creates a censor function that will replace vowels with _.
   - Then `(censor-vowels "Racket is fun")` should produce `"R_ck_t _s f_n"` (all lowercase vowels are replaced by _, while uppercase letters and other characters remain unchanged).
   - Similarly, `((make-censor "123") "c3p0 is 1337")` should yield `"c_p0 is ___7"`, replacing any `1`, `2`, or `3` with _.
   - If the forbidden-character string is empty, the returned function should simply return the original string unchanged (since there's nothing to censor).

   Hints: You may find it useful to iterate over the characters of the input string by converting the string to a list of characters (using `string→list`) and back (`list→string`), or by using recursion over the string's characters. Consider using `map` with a lambda to transform each character. You might also design a helper predicate to check if a character is in the forbidden set. Ensure that you use a lambda in creating the returned function.

6. **Negating a Predicate.** Define a function `negate` that takes one argument (a unary predicate function `p`) and returns a new predicate function. The returned function should take one input and return #t if and only if `p` returns #f on that input (and vice versa). In other words, it returns #t whenever `p` returns #f, and #f whenever `p` returns #t. Use a lambda to construct the returned predicate.

   - *Example:* If `even?` is a predicate that returns #t for even numbers and #f for odd numbers, then `(negate even?)` returns a predicate that returns #t for odd numbers and #f for even numbers. For instance, `((negate even?) 4)` yields #f (since 4 is even, the negated predicate returns false), and `((negate even?) 5)` yields #t.
   - *Example:* If `p` is a predicate that checks whether a string is empty, then `(negate p)` will return #t exactly when the string is non-empty (and #f when it is empty).

7. **Making a Suffix Adder.** Define a function `make-suffix` that takes one argument (a string s) and returns a new function. The returned function should accept a single string and return a new string with s appended to the end. In other words, `make-suffix` produces a function that adds a fixed suffix to any given string. Use an anonymous function (lambda) in your implementation.

   - *Example:* `(make-suffix "ly")` produces a function that adds "ly" to a word. If we define `add-ly` as `(make-suffix "ly")`, then `(add-ly "quick")` returns `"quickly"`.
   - *Example:* Let `exclaim` be `(make-suffix "!")`. Then `(exclaim "Wow")` produces `"Wow!"`.